

Datenstrukturen und Algorithmen

Vorlesung 11: Rot-Schwarz-Bäume (K13)

Prof. Dr. Erika Ábrahám

Theorie Hybrider Systeme
Informatik 2

<http://ths.rwth-aachen.de/teaching/ss-14/datenstrukturen-und-algorithmen/>

Diese Präsentation verwendet in Teilen Folien von Joost-Pieter Katoen.

30. Mai 2014



Übersicht

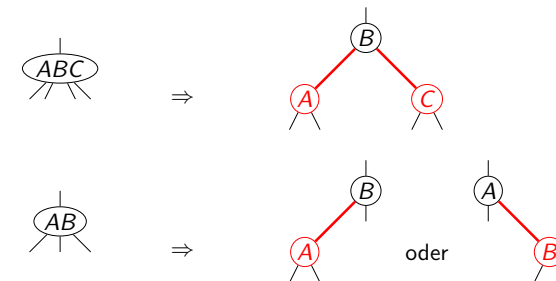
1 Rot-Schwarz-Bäume

Übersicht

1 Rot-Schwarz-Bäume

Motivation

Jeder 2-3-4-Baum kann zu einem Binärbaum (**Rot-Schwarz-Baum**, **red-black-tree**, **RBT**) aufgesplittet werden.

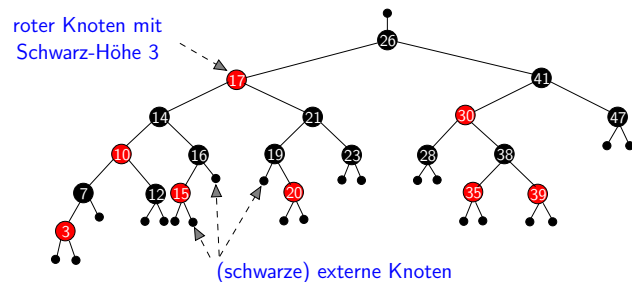


- ▶ Die Anzahl der schwarzen Knoten auf jedem Pfad im RBT entspricht der Anzahl der Knoten auf dem Originalpfad.
- ▶ Vollständigkeit des 2-3-4-Baumes impliziert **farben-balanciertheit** für den RBT: **Die Anzahl schwarzer Knoten ist auf allen Pfaden von der Wurzel zu einem Blatt gleich.**

Motivation

- ▶ Schwächere Balanciertheitsbedingung als für AVL-Bäume, garantiert aber immer noch logarithmische Komplexität für Zugriffsoperationen.
- ▶ Einfacher zu implementieren.
- ▶ Mehr dazu später...

Rot-Schwarz-Bäume



Definition

- ▶ Die **Schwarz-Höhe** $bh(x)$ eines Knotens x ist die Anzahl schwarzer Knoten bis zu einem (externen) Blatt, x ausgenommen.
- ▶ Die **Schwarzhöhe** $bh(t)$ eines RBT t ist die Schwarz-Höhe seiner Wurzel.
- ▶ Die Schwarz-Höhe eines externen Blattes $bh(\text{null}) = 0$.

Rot-Schwarz-Bäume

Rot-Schwarz-Eigenschaft

Ein binärer Suchbaum, dessen Knoten jeweils zusätzlich eine Farbe haben, hat die **Rot-Schwarz-Eigenschaft**, wenn:

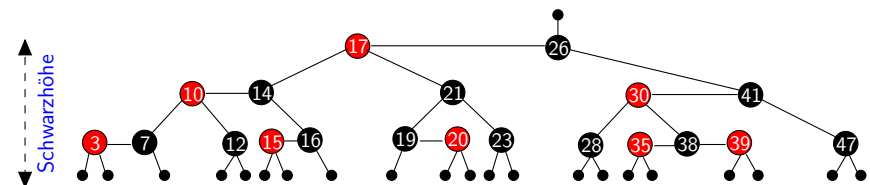
1. Jeder Knoten ist entweder rot oder schwarz.
2. Die Wurzel ist schwarz.
3. Ein roter Knoten hat nur schwarze Kinder.
4. `null`-Zeiger (fehlendes Kind, hier enden die Pfade) betrachten wir als **externe Knoten** mit der Farbe schwarz.
5. Für jeden Knoten enthalten alle Pfade, die an diesem Knoten starten und in einem externen Knoten enden, die gleiche Anzahl schwarzer Knoten.

Solche Bäume heißen dann **Rot-Schwarz-Bäume** (red-black tree, RBT).

- ▶ In den Algorithmen verwenden wir für `null`-Zeiger (externe Knoten) die Notation `null.color == BLACK`.

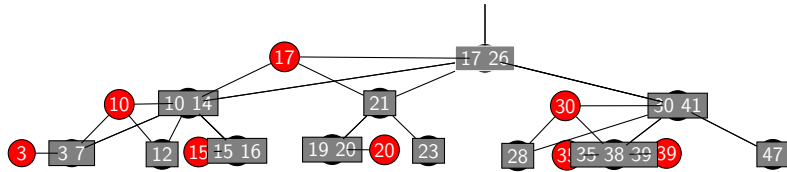
Rot-Schwarz-Bäume

Zeichnet man die **roten** Knoten auf der selben Höhe wie ihren Vater, dann erhält man:



- ▶ Die externen Knoten werden in Zeichnungen oft weggelassen.

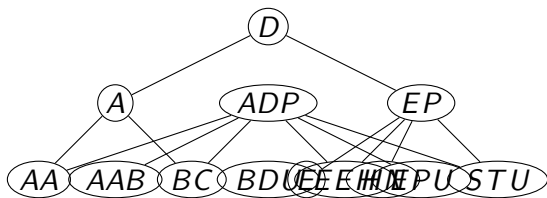
Zurückführung in 2-3-4-Bäume



2-3-4-Bäume: Einfügen eines Schlüssels

- ▶ **Steige** von der Wurzel zum Blatt, in dem eingefügt werden soll, hinunter (ähnlich wie im Binärbaum).
- ▶ **Spalte** auf dem Weg alle maximalen Knoten auf (beim Spalten einer maximalen Wurzel wird der Baum nach oben erhöht).
- ▶ **Füge** den Schlüssel im Blatt ein.

Beispiel: Insert "BUDAPEST" und "AACHEN" in einen leeren 2-3-4-Baum



Elementare Eigenschaften von Rot-Schwarz-Bäumen

Lemma

Ein Rot-Schwarz-Baum t mit Schwarzhöhe $h = bh(t)$ hat:

- ▶ Mindestens $2^h - 1$ innere Knoten.
- ▶ Höchstens $4^h - 1$ innere Knoten.

Beweis: Induktion über h .

Theorem

Ein RBT mit n inneren Knoten hat höchstens die Höhe $2 \cdot \log(n + 1)$.

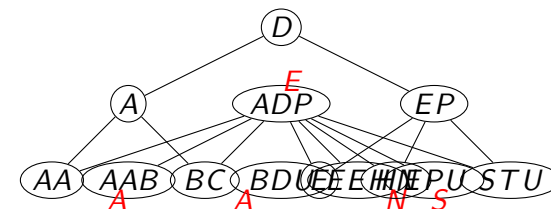
- ⇒ Suchen benötigt also nur $\Theta(\log n)$ statt $\Theta(n)$ Zeit.
- ▶ Für `bstMin`, `bstSucc`, etc. gilt dasselbe.
- ▶ Wir beschäftigen uns im Folgenden mit Einfügen und Löschen.

2-3-4-Bäume: Einfügen eines Schlüssels

In RBT wird genauso verfahren, aber die 4-Knoten werden nicht auf dem Weg nach unten gespalten, sondern nach dem Einfügen rekursiv nach oben.

- ▶ **Steige** von der Wurzel zum Blatt, in dem eingefügt werden soll, hinunter.
- ▶ **Füge** den Schlüssel im Blatt ein.
- ▶ **Spalte** rückwärts eventuelle "5-Knoten" auf.

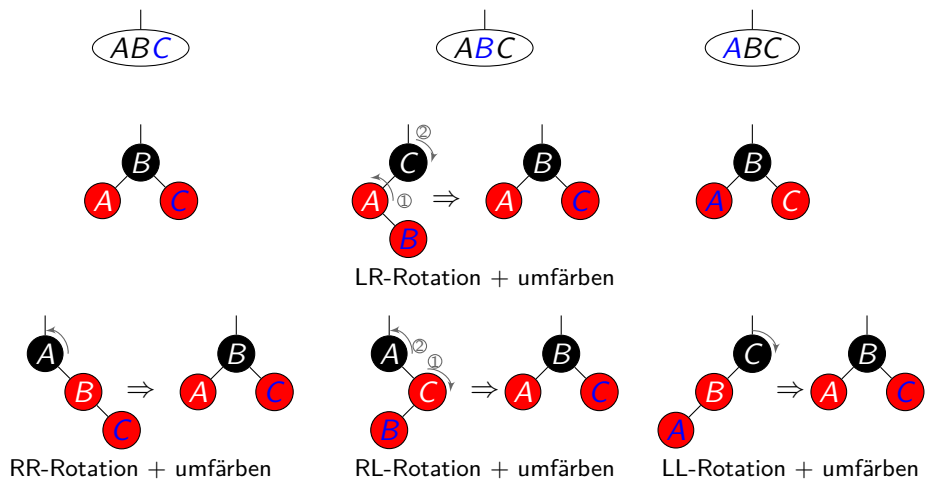
Beispiel: Insert "BUDAPEST" und "AACHEN" in einen leeren 2-3-4-Baum



Wir brauchen für die RBT Implementierung...

- ▶ Einfügen eines Schlüssels in ein Blatt
- ▶ Split-Operation für 4-Knoten

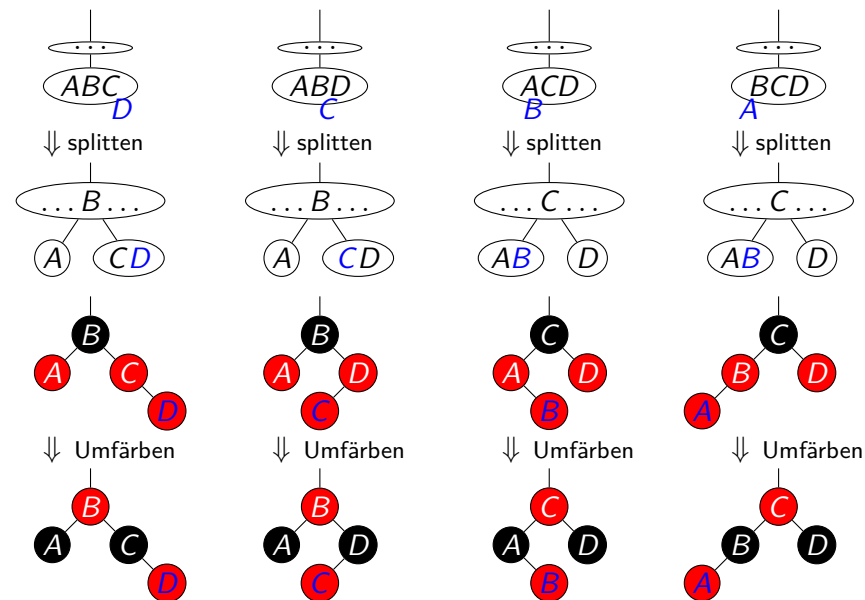
Einfügen in Blatt mit 2 Schlüssel



Einfügen in Blatt mit 1 Schlüssel



Einfügen in Blatt mit 3 Schlüssel



Einfügen von Schlüssel k in einen RBT – Strategie

Einfügen

Zum Einfügen in einen RBT gehen wir zunächst wie beim BST vor:

- ▶ **Finde** einen geeigneten, freien Platz.
- ▶ **Hänge** den neuen Knoten **an**.

Es bleibt die Frage nach der **Farbe**:

- ▶ Färben wir den neuen Knoten schwarz, dann verletzen wir in der Regel die Schwarz-Höhen-Bedingung.
- ▶ Färben wir ihn aber **rot**, dann könnten wir eine Verletzung der Farbbedingungen bekommen (die Wurzel ist schwarz, rote Knoten haben keine roten Kinder).

⇒ Wir färben den Knoten **rot** – ein Schwarz-Höhen-Verletzung wäre schwieriger zu behandeln.

- ▶ **Rot** ist sozusagen eine *lokale* Eigenschaft, schwarz eine *globale*.
- ▶ **Behebe** daher im letzten Schritt die Farbverletzung.

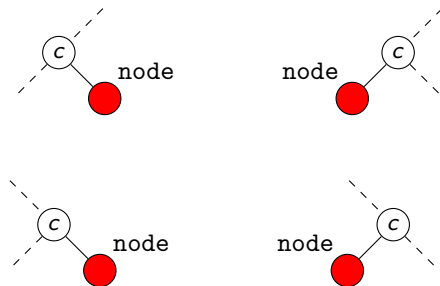
Einfügen in einen RBT – Algorithmus

```

1 void rbtIns(Tree t, Node node) { // Füge node in den Baum t ein
2   bstIns(t, node); // Einfügen wie beim BST
3   node.left = null;
4   node.right = null;
5   node.color = RED; // eingefügter Knoten immer zunächst rot
6   // stelle Rot-Schwarz-Eigenschaft ggf. wieder her
7   rbtInsFix(t, node);
8 }

```

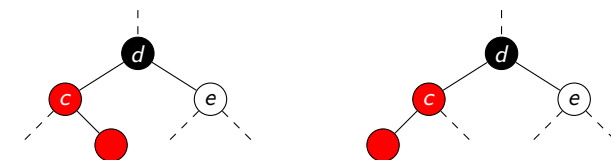
Einfügen – Was kann passieren?



- ▶ Der neu eingefügte Knoten ist immer **rot**.
- ▶ Ist die Farbe des Vaterknotens c schwarz (z. B. die Wurzel), haben wir kein Problem (**2-3-4-Vater hatte 1 oder 2 Schlüssel**).
- ▶ Ist c aber **rot**, dann liegt eine **Rot-Rot-Verletzung** vor, die wir behandeln müssen.
- ▶ Die unteren Fälle lassen sich analog (symmetrisch) zu den oberen lösen, daher betrachten wir nur die beiden oberen Situationen.

Einfügen – Was kann passieren?

Wir müssen nun Großvater d und Onkel e mit berücksichtigen:



- ▶ Der Großvater des eingefügten Knotens war schwarz, denn es handelte sich vor dem Einfügen um einen korrekten RBT.

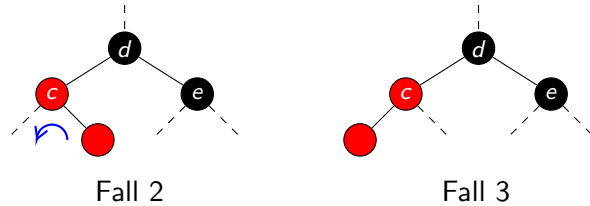
Fall 1

Ist e **rot**, dann können wir durch Umfärben von c und e auf schwarz sowie d auf **rot** die Rot-Schwarz-Eigenschaft lokal wieder herstellen (**2-3-4-Vater hatte 3 Schlüssel, splitten**).

- ▶ **Zwei Ebenen** weiter oben könnte nun aber eine **Rot-Rot-Verletzung** vorliegen, die nach dem selben Schema **iterativ** aufgelöst werden kann.
- ▶ Ist d allerdings die **Wurzel**, dann färben wir sie einfach wieder schwarz. Dadurch erhöht sich die Schwarzhöhe des Baumes um 1.

Einfügen – Was kann passieren?

Ist der Onkel e dagegen schwarz (2-3-4-Vater hatte 2 Schlüsseln), dann erhalten wir Fall 2 und 3:



Fall 2

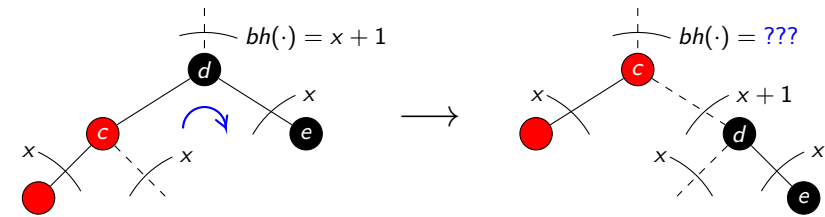
Fall 3

Fall 2

Dieser Fall lässt sich durch Linksrotation um c auf Fall 3 reduzieren.

- ▶ Die Schwarz-Höhe des linken Teilbaumes von d ändert sich dadurch nicht.
- ▶ Der bisherige Vaterknoten c wird dabei zum linken, roten Kind, das eine Rot-Rot-Verletzung mit dem neuen Vater (c im rechten Bild) hat, die wir mit Fall 3 beheben können.

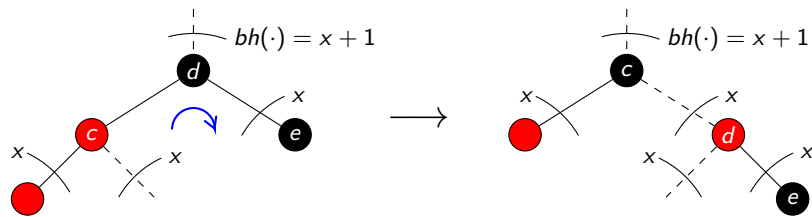
Einfügen – Was kann passieren?



Fall 3

- ▶ Zunächst rotieren wir um d nach rechts, wobei wir die Schwarz-Höhen im Auge behalten.
- ▶ Um die Schwarz-Höhen wieder in Einklang zu bringen, färben wir d rot und c schwarz.

Einfügen – Was kann passieren?



Fall 3

- ▶ Zunächst rotieren wir um d nach rechts, wobei wir die Schwarz-Höhen im Auge behalten.
- ▶ Um die Schwarz-Höhen wieder in Einklang zu bringen, färben wir d rot und c schwarz.
- ▶ Somit haben wir wieder einen gültigen RBT (mit max. zwei Rotationen). Die Schwarzhöhe des Gesamtbaumes ist unverändert.

Einfügen in einen RBT – Algorithmus Teil 2

```

1 // Behebe eventuelle Rot-Rot-Verletzung mit Vater, node ist rot
2 void rbtInsFix(Tree t, Node node) {
3 // solange noch eine Rot-Rot-Verletzung besteht
4 // NB: root.parent.color==black
5 while (node.parent.color == RED) {
6     if (node.parent == node.parent.parent.left) {
7         // der von uns betrachtete Fall
8         node = leftAdjust(t, node); // node jetzt weiter oben?
9         // (node = node.parent.parent im Fall 1 von leftAdjust)
10    } else {
11        // der dazu symmetrischer Fall
12        node = rightAdjust(t, node);
13    }
14 }
15 t.root.color = BLACK; // Wurzel bleibt schwarz
16 }

```

Einfügen in einen RBT – Algorithmus Teil 3

```

1 Node leftAdjust(Tree t, Node node) {
2   Node uncle = node.parent.parent.right;
3   if (uncle.color == RED) { // Fall 1: Split
4     node.parent.parent.color = RED; // Großvater
5     node.parent.color = BLACK; // Vater
6     uncle.color = BLACK; // Onkel
7     return node.parent.parent; // prüfe Rot-Rot weiter oben
8   } else { // Fall 2 und 3
9     if (node == node.parent.right) { // Fall 2
10      // dieser Knoten wird das linke, rote Kind:
11      node = node.parent;
12      leftRotate(t, node);
13    } // Fall 3
14    rightRotate(t, node.parent.parent);
15    node.parent.color = BLACK;
16    node.parent.right.color = RED;
17    return node; // fertig, node.parent.color == BLACK
18  }
19 }

```

Einfügen in einen RBT – Analyse

Zeitkomplexität Einfügen

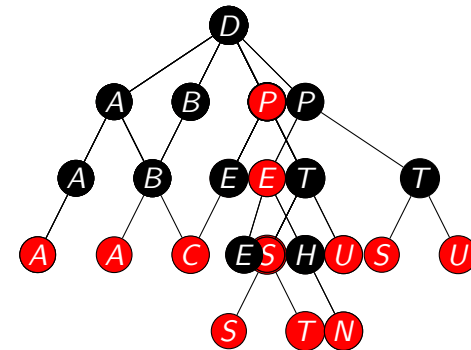
Die Worst-Case Laufzeit von `rbtIns` für ein RBT mit n inneren Knoten ist $O(\log n)$.

Beweisskizze:

- ▶ Die Worst-Case Laufzeit von `bstIns` ist $O(\log n)$.
 - ▶ Die Schleife in `rbtInsFix` wird nur wiederholt wenn Fall 1 auftritt. Dann steigt der Zeiger `node` zwei Ebenen im Baum auf.
 - ▶ Die maximal Anzahl der Schleifen ist damit $O(\log n)$.
 - ▶ Maximal 2 Rotationen werden ausgeführt, da die Schleife in `rbtInsFix` terminiert, wenn die Fälle 2 oder 3 auftreten. (Fall 1 involviert keine Rotationen.)
- ⇒ Die Gesamtanzahl der Rotationen ist konstant und eine Rotation läuft in $O(1)$.
- ▶ Somit benötigt `rbtIns` eine Gesamtzeit $O(\log n)$.

Beispiel RBT: Einfügen eines Schlüssels

Beispiel: Insert "BUDAPEST" und "AACHEN" in einen leeren RBT



Erinnerung: Löschen im 2-3-4-Baum

1. **Steige** von der Wurzel zum Knoten (nicht unbedingt Blatt), in dem der Schlüssel k gelöscht werden soll, hinunter (ähnlich zum Einfügen).
2. **Vergrößere** auf dem Weg alle minimalen Knoten durch
 - 2.1 **Verschiebung** von Schlüssel oder
 - 2.2 **Verschmelzung** von Knoten.
3. Fallunterscheidung Zielknoten:
 - 3.1 **Blatt**: lösche den Schlüssel k (beachte: Blatt ist nicht minimal!).
 - 3.2 **Innerer Knoten**:
 - 3.2.1 Wenn ein Kind, das entweder den Nachfolger- oder den Vorgängerschlüssel k' von k enthält, nicht minimal ist, dann lösche k' (rekursiv) und ersetze k durch k' .
 - 3.2.2 Sonst verschmelze die Kinder, die den Vorgänger- und Nachfolgerschlüsseln enthalten, mit k und lösche k rekursiv.

Wie das Splitten während des Einfügens, für RBTs machen wir auch das Vergrößern nicht auf dem Weg nach unten, sondern nach Erreichen des zu löschenden Schlüssels nach oben.

Löschen im RBT – Strategie

Löschen

Damit der RBT auch nach dem Löschen noch ein RBT ist, müssen wir das Löschverfahren für BSTs ergänzen:

- ▶ Werden wir einen **rot**en Knoten (wirklich) löschen, bleibt alles beim alten, da:
 1. im Baum werden keine Schwarzhöhen geändert
 2. es entstehen keine benachbarten **rot**en Knoten
 3. der dann gelöschte Knoten war rot, also bleibt die Wurzel schwarz.

⇒ Also: das Löschen eines roten Knotens liefert ein RBT.

- ▶ Wollen wir dagegen einen schwarzen Knoten löschen, dann haben wir auf dem Pfad von der Wurzel zum gelöschten Knoten, bzw. zum an seine Stelle getretenen Knoten einen Schwarzwert zu viel.

⇒ **Behebe** diese Schwarz-Höhen-Verletzung.

Löschen im RBT – Algorithmus Teil 1

```

1 // Entfernt node aus dem Baum.
2 void rbtDel(Tree t, Node node) {
3   if (node.left && node.right) {
4     // zwei Kinder: ersetze Knoten durch den Nachfolger
5     Node tmp = bstMin(node.right); // finde den Nachfolger
6     rbtDel(t, tmp); // lösche den Nachfolger
7     bstSwap(t, node, tmp); // ersetze node durch Nachfolger
8     tmp.color = node.color; // übernehme die Farbe
9   } else { // zu löschender Schlüssel ist in 2- oder 3-Blatt
10    Node child;
11    if (node.left) child = node.left; // Kind ist links
12    else if (node.right) child = node.right; // Kind ist rechts
13    else child = null; // kein Kind
14    rbtDelFix(t, node, child);
15    bstReplace(t, node, child);
16  }
17 }
```

Löschen im RBT – Strategie

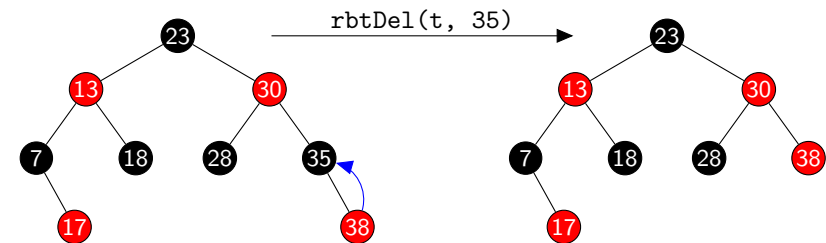
Löschen

Insbesondere für den Fall `node` hat zwei Kinder:

1. Wir finden den **Nachfolger** von `node`
2. Entfernen ihn (mittels `rbtDel`) aus seiner ursprünglichen Position
3. Und **beheben dabei die möglich auftretende Farbverletzung**
4. Ersetzen `node` durch den Nachfolger, und
5. Übernehmen dabei die **möglicherweise neue** Farbe von `node`.

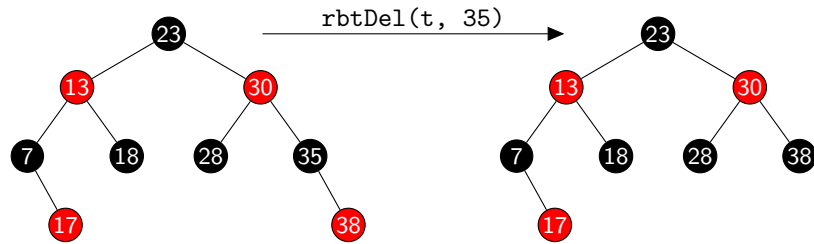
Dadurch ändert sich *nichts mehr* an den Farben, der *RBT bleibt gültig!*

Löschen im RBT – Beispiel 1



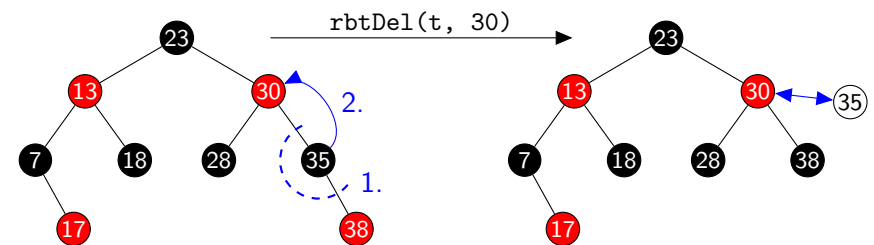
- ▶ Wie beim BST tritt der rechte Teilbaum von 35 an dessen Stelle.
- ▶ Da einer der beiden Knoten rot ist, kann vorher die Farbe der Knoten einfach „vertauscht“ werden; wir färben dazu 38 schwarz (35 wird sowieso gelöscht).
- ▶ Wäre auch 38 bereits schwarz gewesen, hätten wir die Verletzung aufwändiger weiter oben beheben müssen, indem 35 seinen Schwarzwert in Richtung Wurzel „weitergibt“.

Löschen im RBT – Beispiel 1



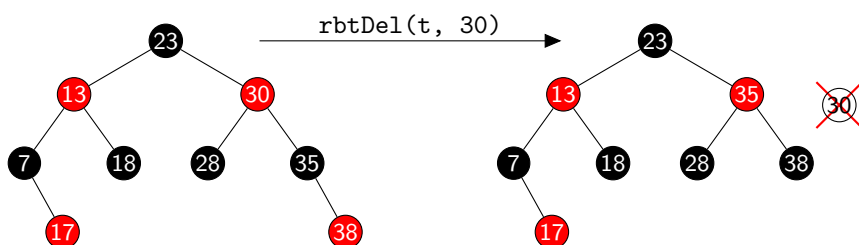
- ▶ Wie beim BST tritt der rechte Teilbaum von 35 an dessen Stelle.
- ▶ Da einer der beiden Knoten rot ist, kann vorher die Farbe der Knoten einfach „vertauscht“ werden; wir färben dazu 38 schwarz (35 wird sowieso gelöscht).
- ▶ Wäre auch 38 bereits schwarz gewesen, hätten wir die Verletzung aufwändiger weiter oben beheben müssen, indem 35 seinen Schwarzwert in Richtung Wurzel „weitergibt“.

Löschen im RBT – Beispiel 2



- ▶ Da 30 zwei Kinder hat, finde den Nachfolger.
- ▶ Lösche 35 so, dass die RBT-Eigenschaft erhalten bleibt (siehe voriges Beispiel).
- ▶ Ersetze 30 durch die nun freie 35, wobei die Farbe von 30 übernommen wird.

Löschen im RBT – Beispiel 2



- ▶ Da 30 zwei Kinder hat, finde den Nachfolger.
- ▶ Lösche 35 so, dass die RBT-Eigenschaft erhalten bleibt (siehe voriges Beispiel).
- ▶ Ersetze 30 durch die nun freie 35, wobei die Farbe von 30 übernommen wird.

Löschen im RBT – Algorithmus Teil 2

```

1 // node soll gelöscht werden, child ist das einzige Kind
2 // (bzw. node hat keine Kinder, dann ist child == node);
3 // ist node rot, so ist nichts zu tun; sonst suchen wir
4 // einen roten Knoten, der durch Umfärben auf schwarz
5 // die schwarze Farbe von node übernimmt
6 void rbtDelFix(Tree t, Node node, Node child) {
7     if (node.color == RED) return;
8     if (child.color == RED) {
9         child.color = BLACK;
10    } else {
11        Node searchPos = node;
12        // solange der Schwarzwert nicht eingefügt werden kann
13        while (searchPos.parent && searchPos.color == BLACK) {
14            if (searchPos == searchPos.parent.left) //linkes Kind
15                searchPos = delLeftAdjust(t, searchPos);
16            else // rechtes Kind
17                searchPos = delRightAdjust(t, searchPos);
18        }
19        searchPos.color=BLACK;
20    }
21 }

```

Löschen im RBT – Algorithmus Teil 3a

```

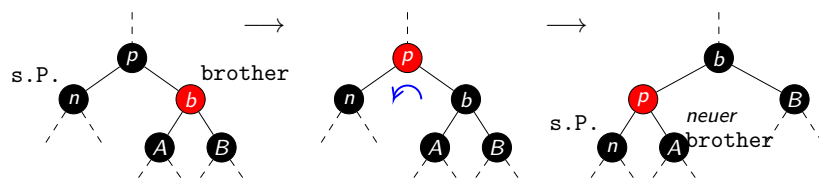
1 // Erleichtert searchPos um einen Schwarzwert,
2 // wobei searchPos das linke Kind ist.
3 Node delLeftAdjust(Tree t, Node searchPos) {
4 // brother existiert immer wegen Schwarzhöhe
5 Node brother = searchPos.parent.right;
6 if (brother.color == RED) {
7 // Fall 1: Reduktion auf 2,3,4
8 // NB: Vater ist schwarz, da Bruder rot
9 brother.color = BLACK;
10 searchPos.parent.color = RED; // Vater
11 leftRotate(t, searchPos.parent);
12 brother = searchPos.parent.right; // nun Bruder von node
13 }

```

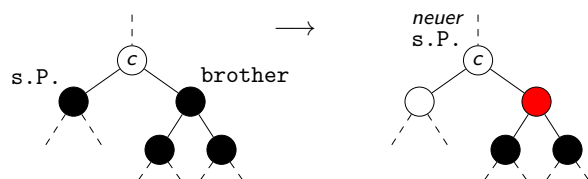
→

Der Löschalgorithmus – Fall 1 und 2 (s.P. steht für searchPos)

Fall 1: (Reduktion auf Fall 2, 3 oder 4)



Fall 2:



Löschen im RBT – Algorithmus Teil 3b

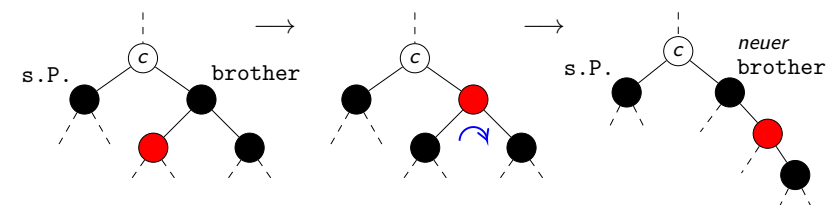
```

12 if (brother.left.color == BLACK &&
13     brother.right.color == BLACK) { // Fall 2
14     brother.color = RED;
15     return searchPos.parent; // Doppel-schwarz weiter oben...
16 } else { // Fall 3 und 4
17     if (brother.right.color == BLACK) // Fall 3
18         brother.left.color = BLACK;
19         brother.color = RED;
20         rightRotate(t, brother);
21         brother = searchPos.parent.right; // neuer Bruder
22     } // Fall 4
23     brother.color = searchPos.parent.color;
24     searchPos.parent.color = BLACK;
25     brother.right.color = BLACK;
26     leftRotate(t, searchPos.parent);
27     return t.root; // Fertig.
28 }
29 }

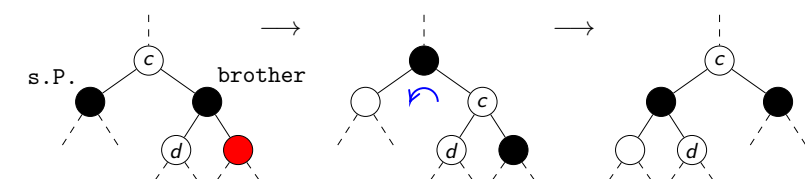
```

Der Löschalgorithmus – Fall 3 und 4 (s.P. steht für searchPos)

Fall 3: (Reduktion auf Fall 4)



Fall 4:



Löschen im RBT – Analyse

Zeitkomplexität Löschen

Die Worst-Case Laufzeit von `rbtDel` für ein RBT mit n inneren Knoten ist $O(\log n)$.

Beweisskizze:

- ▶ Die Laufzeit von `rbtDel` ohne Aufrufe von `rbtDelFix` ist $O(\log n)$.
- ▶ Die Fälle 2, 3, und 4 brauchen höchstens 3 Rotationen und eine konstante Anzahl Farbänderungen.
- ▶ Die Schleife in `rbtDelFix` wird nur wiederholt wenn Fall 2 auftritt. Dann steigt der Zeiger `node` eine Ebene im Baum auf.
- ▶ Die maximal Anzahl der Schleifen ist damit $O(\log n)$.
- ▶ Somit benötigt `rbtDel` eine Gesamtzeit $O(\log n)$.

Komplexität der RBT-Operationen

Operation	Zeit
<code>bstSearch</code>	$\Theta(h)$
<code>bstSucc</code>	$\Theta(h)$
<code>bstMin</code>	$\Theta(h)$
<code>bstIns</code>	$\Theta(h)$
<code>bstDel</code>	$\Theta(h)$

Operation	Zeit
<code>rbtIns</code>	$\Theta(\log n)$
<code>rbtDel</code>	$\Theta(\log n)$

- ▶ Alle anderen Operationen wie beim BST, wobei $h = \log n$.

Alle Operationen sind logarithmisch in der Größe des Rot-Schwarz-Baumes